**QuickStart**

Good tools make application development quicker and easier to maintain than if you did everything by hand.

The [Angular CLI](https://cli.angular.io/) is a *command line interface* tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.The goal in this guide is to build and run a simple Angular application in TypeScript, using the Angular CLI while adhering to the [Style Guide](https://angular.io/guide/styleguide)  recommendations that benefit *every* Angular project.

By the end of the chapter, you'll have a basic understanding of development with the CLI and a foundation for both these documentation samples and for real world applications.

Step 1. Set up the Development Environment

You need to set up your development environment before you can do anything.

Install [Node.js® and npm](https://nodejs.org/en/download/) if they are not already on your machine.

Verify that you are running at least Node.js version 8.x or greater and npm version 5.x or greater by running node -v and npm -v in a terminal/console window. Older versions produce errors, but newer versions are fine. Then install the [Angular CLI](https://github.com/angular/angular-cli) globally.

content\_copy npm install -g @angular/cli

Step 2. Create a new project

Open a terminal window.

Generate a new project and default app by running the following command:

content\_copyng new my-app

The Angular CLI installs the necessary npm packages, creates the project files, and populates the project with a simple default app. This can take some time.

You can add pre-packaged functionality to a new project by using the ng add command. The ng addcommand transforms a project by applying the schematics in the specified package. For more information, see the [Angular CLI documentation.](https://github.com/angular/angular-cli/wiki/add)

Angular Material provides schematics for typical app layouts. See the [Angular Material documentation](https://material.angular.io/guides)for details.

Step 3: Serve the application

Go to the project directory and launch the server.

content\_copycd my-app

ng serve --open

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

Using the --open (or just -o) option will automatically open your browser on http://localhost:4200/.

Step 4: Edit your first Angular component

The CLI created the first Angular component for you. This is the *root component* and it is named app-root. You can find it in ./src/app/app.component.ts.

Open the component file and change the title property from 'app' to 'My First Angular App!'.

src/app/app.component.ts

content\_copyexport class AppComponent {

title = 'My First Angular App!';

}

The browser reloads automatically with the revised title. That's nice, but it could look better.

Open src/app/app.component.css and give the component some style.

src/app/app.component.css

content\_copyh1 {

color: #369;

font-family: Arial, Helvetica, sans-serif;

font-size: 250%;

}

Looking good!

What's next?

That's about all you'd expect to do in a "Hello, World" app.

You're ready to take the [Tour of Heroes Tutorial](https://angular.io/tutorial) and build a small application that demonstrates the great things you can build with Angular.

|  |  |
| --- | --- |
| **File** | **Purpose** |
| app/app.component.{ts,html,css,spec.ts} | Defines the AppComponent along with an HTML template, CSS stylesheet, and a unit test. It is the root component of what will become a tree of nested components as the application evolves. |
| app/app.module.ts | [Defines AppModule, the root module that tells Angular how to assemble the application. Right now it declares only the AppComponent. Soon there will be more components to declare.](https://angular.io/guide/bootstrapping) |
| assets/\* | A folder where you can put images and anything else to be copied wholesale when you build your application. |
| environments/\* | This folder contains one file for each of your destination environments, each exporting simple configuration variables to use in your application. The files are replaced on-the-fly when you build your app. You might use a different API endpoint for development than you do for production or maybe different analytics tokens. You might even use some mock services. Either way, the CLI has you covered. |
| browserslist | [A configuration file to share target browsers between different front-end tools.](https://github.com/browserslist/browserslist) |
| favicon.ico | Every site wants to look good on the bookmark bar. Get started with your very own Angular icon. |
| index.html | The main HTML page that is served when someone visits your site. Most of the time you'll never need to edit it. The CLI automatically adds all js and css files when building your app so you never need to add any <script> or <link> tags here manually. |
| karma.conf.js | [Unit test configuration for the Karma test runner, used when running ng test.](https://karma-runner.github.io/) |
| main.ts | The main entry point for your app. Compiles the application with the JIT compilerand bootstraps the application's root module (AppModule) to run in the browser. You can also use the AOT compiler without changing any code by appending the--aotflag to the ng build and ng serve commands. |
| polyfills.ts | [Different browsers have different levels of support of the web standards. Polyfills help normalize those differences. You should be pretty safe with core-js and zone.js, but be sure to check out the Browser Support guide for more information.](https://angular.io/guide/browser-support) |
| styles.css | Your global styles go here. Most of the time you'll want to have local styles in your components for easier maintenance, but styles that affect all of your app need to be in a central place. |
| test.ts | This is the main entry point for your unit tests. It has some custom configuration that might be unfamiliar, but it's not something you'll need to edit. |
| tsconfig.{app|spec}.json | TypeScript compiler configuration for the Angular app (tsconfig.app.json) and for the unit tests (tsconfig.spec.json). |
| tslint.json | Additional Linting configuration for TSLint together with Codelyzer, used when running ng lint. Linting helps keep your code style consistent. |

The root folder

The src/ folder is just one of the items inside the project's root folder. Other files help you build, test, maintain, document, and deploy the app. These files go in the root folder next to src/.

my-app

e2e

src

app.e2e-spec.ts

app.po.ts

tsconfig.e2e.json

protractor.conf.js

node\_modules/...

src/...

karma.conf.js

.editorconfig

.gitignore

angular.json

package.json

README.md

tsconfig.json

tslint.json

|  |  |
| --- | --- |
| **File** | **Purpose** |
| e2e/ | Inside e2e/ live the end-to-end tests. They shouldn't be inside src/ because e2e tests are really a separate app that just so happens to test your main app. That's also why they have their own tsconfig.e2e.json. |
| node\_modules/ | Node.js creates this folder and puts all third party modules listed in package.jsoninside of it. |
| .editorconfig | [Simple configuration for your editor to make sure everyone that uses your project has the same basic configuration. Most editors support an .editorconfig file. See http://editorconfig.org for more information.](http://editorconfig.org/) |
| .gitignore | Git configuration to make sure autogenerated files are not committed to source control. |
| angular.json | Configuration for Angular CLI. In this file you can set several defaults and also configure what files are included when your project is built. Check out the official documentation if you want to know more. |
| package.json | [npm configuration listing the third party packages your project uses. You can also add your own custom scripts here.](https://docs.npmjs.com/misc/scripts) |
| protractor.conf.js | [End-to-end test configuration for Protractor, used when running ng e2e.](http://www.protractortest.org/) |
| README.md | Basic documentation for your project, pre-filled with CLI command information. Make sure to enhance it with project documentation so that anyone checking out the repo can build your app! |
| tsconfig.json | TypeScript compiler configuration for your IDE to pick up and give you helpful tooling. |
| tslint.json | Linting configuration for TSLint together with Codelyzer, used when running ng lint. Linting helps keep your code style consistent. |

# Tutorial: Tour of Heroes

The Tour of Heroes tutorial covers the fundamentals of Angular.  
In this tutorial you will build an app that helps a staffing agency manage its stable of heroes.

This basic app has many of the features you'd expect to find in a data-driven application. It acquires and displays a list of heroes, edits a selected hero's detail, and navigates among different views of heroic data.

By the end of the tutorial you will be able to do the following:

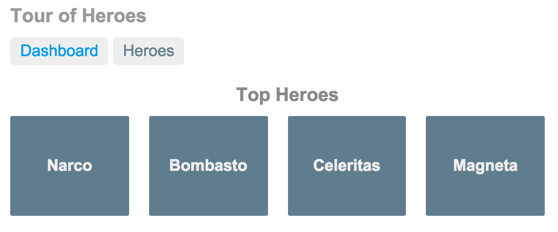
* Use built-in Angular directives to show and hide elements and display lists of hero data.
* Create Angular components to display hero details and show an array of heroes.
* Use one-way data binding for read-only data.
* Add editable fields to update a model with two-way data binding.
* Bind component methods to user events, like keystrokes and clicks.
* Enable users to select a hero from a master list and edit that hero in the details view.
* Format data with pipes.
* Create a shared service to assemble the heroes.
* Use routing to navigate among different views and their components.

You'll learn enough Angular to get started and gain confidence that Angular can do whatever you need it to do.

After completing all tutorial steps, the final app will look like this [live example](https://angular.io/generated/live-examples/toh-pt6/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt6/toh-pt6.zip).

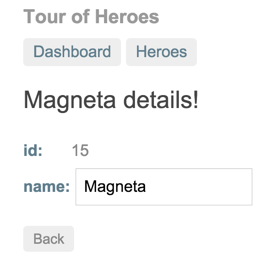
## What you'll build

Here's a visual idea of where this tutorial leads, beginning with the "Dashboard" view and the most heroic heroes:



You can click the two links above the dashboard ("Dashboard" and "Heroes") to navigate between this Dashboard view and a Heroes view.

If you click the dashboard hero "Magneta," the router opens a "Hero Details" view where you can change the hero's name.



Clicking the "Back" button returns you to the Dashboard. Links at the top take you to either of the main views. If you click "Heroes," the app displays the "Heroes" master list view.



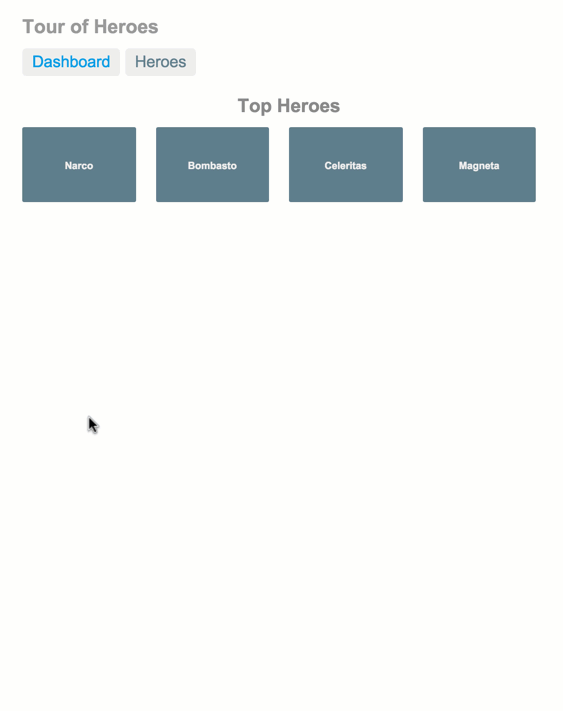
When you click a different hero name, the read-only mini detail beneath the list reflects the new choice.

You can click the "View Details" button to drill into the editable details of the selected hero.

The following diagram captures all of the navigation options.



Here's the app in action:



**The Application Shell**

Install the Angular CLI Install the Angular CLI, if you haven't already done so.

content\_copy

npm install -g @angular/cli

Create a new application Create a new project named angular-tour-of-heroes with this CLI command.

**ng new angular-tour-of-heroes**

The Angular CLI generated a new project with a default application and supporting files.You can add pre-packaged functionality to a new project by using the ng add command. The ng add command transforms a project by applying the schematics in the specified package. For more information, see the Angular CLI documentation.

Angular Material provides schematics for typical app layouts. See the Angular Material documentation for details.

Serve the application Go to the project directory and launch the application.

**cd angular-tour-of-heroes**

**ng serve –open**

The ng serve command builds the app, starts the development server, watches the source files, and rebuilds the app as you make changes to those files.

The --open flag opens a browser to http://localhost:4200/.

You should see the app running in your browser.

**Angular components**

The page you see is the application shell. The shell is controlled by an Angular component named AppComponent.

Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

Change the application title

Open the project in your favorite editor or IDE and navigate to the src/app folder.

You'll find the implementation of the shell AppComponent distributed over three files:

app.component.ts— the component class code, written in TypeScript.

app.component.html— the component template, written in HTML.

app.component.css— the component's private CSS styles.

Open the component class file (app.component.ts) and change the value of the title property to 'Tour of Heroes'.

app.component.ts (class title property)

content\_copy

title = 'Tour of Heroes';

Open the component template file (app.component.html) and delete the default template generated by the Angular CLI. Replace it with the following line of HTML.

app.component.html (template)

content\_copy

<h1>{{title}}</h1>

The double curly braces are Angular's interpolation binding syntax. This interpolation binding presents the component's title property value inside the HTML header tag.

The browser refreshes and displays the new application title.

Add application styles

Most apps strive for a consistent look across the application. The CLI generated an empty styles.css for this purpose. Put your application-wide styles there.

Here's an excerpt from the styles.css for the Tour of Heroes sample app.

src/styles.css (excerpt)

content\_copy

/\* Application-wide Styles \*/

h1 {

color: #369;

font-family: Arial, Helvetica, sans-serif;

font-size: 250%;

}

h2, h3 {

color: #444;

font-family: Arial, Helvetica, sans-serif;

font-weight: lighter;

}

body {

margin: 2em;

}

body, input[text], button {

color: #888;

font-family: Cambria, Georgia;

}

/\* everywhere else \*/

\* {

font-family: Arial, Helvetica, sans-serif;

}

Final code review

The source code for this tutorial and the complete Tour of Heroes global styles are available in the live example / download example.

Here are the code files discussed on this page.

src/app/app.component.ts

src/app/app.component.html

src/styles.css (excerpt)

content\_copy

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Tour of Heroes';

}

Summary

You created the initial application structure using the Angular CLI.

You learned that Angular components display data.

You used the double curly braces of interpolation to display the app title.

**The Hero Editor**

The application now has a basic title. Next you will create a new component to display hero information and place that component in the application shell.

Create the heroes component

Using the Angular CLI, generate a new component named heroes.

**ng generate component heroes**

The CLI creates a new folder, src/app/heroes/ and generates the three files of the HeroesComponent.

The HeroesComponent class file is as follows:

app/heroes/heroes.component.ts (initial version)

**import { Component, OnInit } from '@angular/core';**

**@Component({**

**selector: 'app-heroes',**

**templateUrl: './heroes.component.html',**

**styleUrls: ['./heroes.component.css']**

**})**

**export class HeroesComponent implements OnInit {**

**constructor() { }**

**ngOnInit() {**

**}**

**}**

You always import the Component symbol from the Angular core library and annotate the component class with @Component.

@Component is a decorator function that specifies the Angular metadata for the component.

The CLI generated three metadata properties:

selector— the component's CSS element selector

templateUrl— the location of the component's template file.

styleUrls— the location of the component's private CSS styles.

The CSS element selector, 'app-heroes', matches the name of the HTML element that identifies this component within a parent component's template.

The ngOnInit is a lifecycle hook Angular calls ngOnInit shortly after creating a component. It's a good place to put initialization logic.

Always export the component class so you can import it elsewhere ... like in the AppModule.

Add a hero property Add a hero property to the HeroesComponent for a hero named "Windstorm."

heroes.component.ts (hero property)

**hero = 'Windstorm';**

Show the hero Open the heroes.component.html template file. Delete the default text generated by the Angular CLI and replace it with a data binding to the new hero property.

heroes.component.html

**{{hero}}**

Show the HeroesComponent view

To display the HeroesComponent, you must add it to the template of the shell AppComponent.

Remember that app-heroes is the element selector for the HeroesComponent. So add an <app-heroes> element to the AppComponent template file, just below the title.

src/app/app.component.html

**<h1>{{title}}</h1>**

**<app-heroes></app-heroes>**

Assuming that the CLI ng serve command is still running, the browser should refresh and display both the application title and the hero name.

Create a Hero class A real hero is more than a name.

Create a Hero class in its own file in the src/app folder. Give it id and name properties.

src/app/hero.ts

**export class Hero {**

**id: number;**

**name: string;**

**}**

Return to the HeroesComponent class and import the Hero class.

Refactor the component's hero property to be of type Hero. Initialize it with an id of 1 and the name Windstorm.

The revised HeroesComponent class file should look like this:

src/app/heroes/heroes.component.ts

**import { Component, OnInit } from '@angular/core';**

**import { Hero } from '../hero';**

**@Component({**

**selector: 'app-heroes',**

**templateUrl: './heroes.component.html',**

**styleUrls: ['./heroes.component.css']**

**})**

**export class HeroesComponent implements OnInit {**

**hero: Hero = {**

**id: 1,**

**name: 'Windstorm'**

**};**

**constructor() { }**

**ngOnInit() {**

**}**

**}**

The page no longer displays properly because you changed the hero from a string to an object.

Show the hero object Update the binding in the template to announce the hero's name and show both id and name in a details layout like this:

heroes.component.html (HeroesComponent's template)

**<h2>{{hero.name}} Details</h2>**

**<div><span>id: </span>{{hero.id}}</div>**

**<div><span>name: </span>{{hero.name}}</div>**

The browser refreshes and display's the hero's information.

Format with the UppercasePipe Modify the hero.name binding like this.

**<h2>{{hero.name | uppercase}} Details</h2>**

The browser refreshes and now the hero's name is displayed in capital letters.

The word uppercase in the interpolation binding, right after the pipe operator ( | ), activates the built-in UppercasePipe.

Pipes are a good way to format strings, currency amounts, dates and other display data. Angular ships with several built-in pipes and you can create your own.

Edit the hero

Users should be able to edit the hero name in an <input> textbox.

The textbox should both display the hero's name property and update that property as the user types. That means data flow from the component class out to the screen and from the screen back to the class.

To automate that data flow, setup a two-way data binding between the <input> form element and the hero.name property.

**Two-way binding**

Refactor the details area in the HeroesComponent template so it looks like this:

src/app/heroes/heroes.component.html (HeroesComponent's template)

**<div>**

**<label>name:**

**<input [(ngModel)]="hero.name" placeholder="name">**

**</label>**

**</div>**

**[(ngModel)] is Angular's two-way data binding syntax.**

Here it binds the hero.name property to the HTML textbox so that data can flow in both directions: from the hero.name property to the textbox, and from the textbox back to the hero.name.

The missing FormsModule

Notice that the app stopped working when you added [(ngModel)].

To see the error, open the browser development tools and look in the console for a message like

**Template parse errors:**

**Can't bind to 'ngModel' since it isn't a known property of 'input'.**

**Although ngModel is a valid Angular directive, it isn't available by default.**

It belongs to the optional FormsModule and you must opt-in to using it.

AppModule

Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires. This information is called metadata

Some of the metadata is in the @Component decorators that you added to your component classes. Other critical metadata is in @NgModule decorators.

The most important @NgModule decorator annotates the top-level AppModule class.

The Angular CLI generated an AppModule class in src/app/app.module.ts when it created the project. This is where you opt-in to the FormsModule.

Import FormsModule

Open AppModule (app.module.ts) and import the FormsModule symbol from the @angular/forms library.

app.module.ts (FormsModule symbol import)

**import { FormsModule } from '@angular/forms'; // <-- NgModel lives here**

Then add FormsModule to the @NgModule metadata's imports array, which contains a list of external modules that the app needs.

app.module.ts ( @NgModule imports)

**imports: [**

**BrowserModule,**

**FormsModule**

**],**

When the browser refreshes, the app should work again. You can edit the hero's name and see the changes reflected immediately in the <h2> above the textbox.

Declare HeroesComponent

Every component must be declared in exactly one NgModule.

You didn't declare the HeroesComponent. So why did the application work?

It worked because the Angular CLI declared HeroesComponent in the AppModule when it generated that component.

Open src/app/app.module.ts and find HeroesComponent imported near the top.

**import { HeroesComponent } from './heroes/heroes.component';**

The HeroesComponent is declared in the @NgModule.declarations array.

**declarations: [**

**AppComponent,**

**HeroesComponent**

**],**

Note that AppModule declares both application components, AppComponent and HeroesComponent.

Final code review : Your app should look like this live example / download example. Here are the code files discussed on this page.

src/app/heroes/heroes.component.ts

src/app/heroes/heroes.component.html

src/app/app.module.ts

src/app/app.component.ts

src/app/app.component.html

src/app/hero.ts

content\_copy

import { Component, OnInit } from '@angular/core';

import { Hero } from '../hero';

@Component({

selector: 'app-heroes',

templateUrl: './heroes.component.html',

styleUrls: ['./heroes.component.css']

})

export class HeroesComponent implements OnInit {

hero: Hero = {

id: 1,

name: 'Windstorm'

};

constructor() { }

ngOnInit() { }

}

Summary

You used the CLI to create a second HeroesComponent.

You displayed the HeroesComponent by adding it to the AppComponent shell.

You applied the UppercasePipe to format the name.

You used two-way data binding with the ngModel directive.

You learned about the AppModule.

You imported the FormsModule in the AppModule so that Angular would recognize and apply the ngModel directive.

You learned the importance of declaring components in the AppModule and appreciated that the CLI declared it for you.

Display a Heroes List

In this page, you'll expand the Tour of Heroes app to display a list of heroes, and allow users to select a hero and display the hero's details.

Create mock heroes

You'll need some heroes to display.

Eventually you'll get them from a remote data server. For now, you'll create some mock heroes and pretend they came from the server.

Create a file called mock-heroes.ts in the src/app/ folder. Define a HEROES constant as an array of ten heroes and export it. The file should look like this.

src/app/mock-heroes.ts

content\_copy

import { Hero } from './hero';

export const HEROES: Hero[] = [

{ id: 11, name: 'Mr. Nice' },

{ id: 12, name: 'Narco' },

{ id: 13, name: 'Bombasto' },

{ id: 14, name: 'Celeritas' },

{ id: 15, name: 'Magneta' },

{ id: 16, name: 'RubberMan' },

{ id: 17, name: 'Dynama' },

{ id: 18, name: 'Dr IQ' },

{ id: 19, name: 'Magma' },

{ id: 20, name: 'Tornado' }

];

Displaying heroes

You're about to display the list of heroes at the top of the HeroesComponent.

Open the HeroesComponent class file and import the mock HEROES.

src/app/heroes/heroes.component.ts (import HEROES)

content\_copy

import { HEROES } from '../mock-heroes';

Add a heroes property to the class that exposes these heroes for binding.

content\_copy

heroes = HEROES;

List heroes with \*ngFor

Open the HeroesComponent template file and make the following changes:

Add an <h2> at the top,

Below it add an HTML unordered list (<ul>)

Insert an <li> within the <ul> that displays properties of a hero.

Sprinkle some CSS classes for styling (you'll add the CSS styles shortly).

Make it look like this:

heroes.component.html (heroes template)

content\_copy

<h2>My Heroes</h2>

<ul class="heroes">

<li>

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

</ul>

Now change the <li> to this:

content\_copy

<li \*ngFor="let hero of heroes">

The \*ngFor is Angular's repeater directive. It repeats the host element for each element in a list.

In this example

<li> is the host element

heroes is the list from the HeroesComponent class.

hero holds the current hero object for each iteration through the list.

Don't forget the asterisk (\*) in front of ngFor. It's a critical part of the syntax.

After the browser refreshes, the list of heroes appears.

Style the heroes

The heroes list should be attractive and should respond visually when users hover over and select a hero from the list.

In the first tutorial, you set the basic styles for the entire application in styles.css. That stylesheet didn't include styles for this list of heroes.

You could add more styles to styles.css and keep growing that stylesheet as you add components.

You may prefer instead to define private styles for a specific component and keep everything a component needs— the code, the HTML, and the CSS —together in one place.

This approach makes it easier to re-use the component somewhere else and deliver the component's intended appearance even if the global styles are different.

You define private styles either inline in the @Component.styles array or as stylesheet file(s) identified in the @Component.styleUrls array.

When the CLI generated the HeroesComponent, it created an empty heroes.component.css stylesheet for the HeroesComponent and pointed to it in @Component.styleUrls like this.

src/app/heroes/heroes.component.ts (@Component)

content\_copy

@Component({

selector: 'app-heroes',

templateUrl: './heroes.component.html',

styleUrls: ['./heroes.component.css']

})

Open the heroes.component.css file and paste in the private CSS styles for the HeroesComponent. You'll find them in the final code review at the bottom of this guide.

Styles and stylesheets identified in @Component metadata are scoped to that specific component. The heroes.component.css styles apply only to the HeroesComponent and don't affect the outer HTML or the HTML in any other component.

Master/Detail

When the user clicks a hero in the master list, the component should display the selected hero's details at the bottom of the page.

In this section, you'll listen for the hero item click event and update the hero detail.

Add a click event binding

Add a click event binding to the <li> like this:

heroes.component.html (template excerpt)

content\_copy

<li \*ngFor="let hero of heroes" (click)="onSelect(hero)">

This is an example of Angular's event binding syntax.

The parentheses around click tell Angular to listen for the <li> element's click event. When the user clicks in the <li>, Angular executes the onSelect(hero) expression.

onSelect() is a HeroesComponent method that you're about to write. Angular calls it with the hero object displayed in the clicked <li>, the same hero defined previously in the \*ngFor expression.

Add the click event handler

Rename the component's hero property to selectedHero but don't assign it. There is no selected hero when the application starts.

Add the following onSelect() method, which assigns the clicked hero from the template to the component's selectedHero.

src/app/heroes/heroes.component.ts (onSelect)

content\_copy

selectedHero: Hero;

onSelect(hero: Hero): void {

this.selectedHero = hero;

}

Update the details template

The template still refers to the component's old hero property which no longer exists. Rename hero to selectedHero.

heroes.component.html (selected hero details)

content\_copy

<h2>{{selectedHero.name | uppercase}} Details</h2>

<div><span>id: </span>{{selectedHero.id}}</div>

<div>

<label>name:

<input [(ngModel)]="selectedHero.name" placeholder="name">

</label>

</div>

Hide empty details with \*ngIf

After the browser refreshes, the application is broken.

Open the browser developer tools and look in the console for an error message like this:

content\_copy

HeroesComponent.html:3 ERROR TypeError: Cannot read property 'name' of undefined

Now click one of the list items. The app seems to be working again. The heroes appear in a list and details about the clicked hero appear at the bottom of the page.

What happened?

When the app starts, the selectedHero is undefined by design.

Binding expressions in the template that refer to properties of selectedHero — expressions like {{selectedHero.name}} — must fail because there is no selected hero.

The fix

The component should only display the selected hero details if the selectedHero exists.

Wrap the hero detail HTML in a <div>. Add Angular's \*ngIf directive to the <div> and set it to selectedHero.

Don't forget the asterisk (\*) in front of ngIf. It's a critical part of the syntax.

src/app/heroes/heroes.component.html (\*ngIf)

content\_copy

<div \*ngIf="selectedHero">

<h2>{{selectedHero.name | uppercase}} Details</h2>

<div><span>id: </span>{{selectedHero.id}}</div>

<div>

<label>name:

<input [(ngModel)]="selectedHero.name" placeholder="name">

</label>

</div>

</div>

After the browser refreshes, the list of names reappears. The details area is blank. Click a hero and its details appear.

Why it works

When selectedHero is undefined, the ngIf removes the hero detail from the DOM. There are no selectedHero bindings to worry about.

When the user picks a hero, selectedHero has a value and ngIf puts the hero detail into the DOM.

Style the selected hero

It's difficult to identify the selected hero in the list when all <li> elements look alike.

If the user clicks "Magneta", that hero should render with a distinctive but subtle background color like this:

Selected hero

That selected hero coloring is the work of the .selected CSS class in the styles you added earlier. You just have to apply the .selected class to the <li> when the user clicks it.

The Angular class binding makes it easy to add and remove a CSS class conditionally. Just add [class.some-css-class]="some-condition" to the element you want to style.

Add the following [class.selected] binding to the <li> in the HeroesComponent template:

heroes.component.html (toggle the 'selected' CSS class)

content\_copy

[class.selected]="hero === selectedHero"

When the current row hero is the same as the selectedHero, Angular adds the selected CSS class. When the two heroes are different, Angular removes the class.

The finished <li> looks like this:

heroes.component.html (list item hero)

content\_copy

<li \*ngFor="let hero of heroes"

[class.selected]="hero === selectedHero"

(click)="onSelect(hero)">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

Final code review

Your app should look like this live example / download example.

Here are the code files discussed on this page, including the HeroesComponent styles.

src/app/heroes/heroes.component.ts

src/app/heroes/heroes.component.html

src/app/heroes/heroes.component.css

content\_copy

import { Component, OnInit } from '@angular/core';

import { Hero } from '../hero';

import { HEROES } from '../mock-heroes';

@Component({

selector: 'app-heroes',

templateUrl: './heroes.component.html',

styleUrls: ['./heroes.component.css']

})

export class HeroesComponent implements OnInit {

heroes = HEROES;

selectedHero: Hero;

constructor() { }

ngOnInit() {

}

onSelect(hero: Hero): void {

this.selectedHero = hero;

}

}

Summary

The Tour of Heroes app displays a list of heroes in a Master/Detail view.

The user can select a hero and see that hero's details.

You used \*ngFor to display a list.

You used \*ngIf to conditionally include or exclude a block of HTML.

You can toggle a CSS style class with a class binding.

Master/Detail Components

At the moment, the HeroesComponent displays both the list of heroes and the selected hero's details.

Keeping all features in one component as the application grows will not be maintainable. You'll want to split up large components into smaller sub-components, each focused on a specific task or workflow.

In this page, you'll take the first step in that direction by moving the hero details into a separate, reusable HeroDetailComponent.

The HeroesComponent will only present the list of heroes. The HeroDetailComponent will present details of a selected hero.

Make the HeroDetailComponent

Use the Angular CLI to generate a new component named hero-detail.

content\_copy

ng generate component hero-detail

The command scaffolds the HeroDetailComponent files and declares the component in AppModule.

Write the template

Cut the HTML for the hero detail from the bottom of the HeroesComponent template and paste it over the generated boilerplate in the HeroDetailComponent template.

The pasted HTML refers to a selectedHero. The new HeroDetailComponent can present any hero, not just a selected hero. So replace "selectedHero" with "hero" everywhere in the template.

When you're done, the HeroDetailComponent template should look like this:

src/app/hero-detail/hero-detail.component.html

content\_copy

<div \*ngIf="hero">

<h2>{{hero.name | uppercase}} Details</h2>

<div><span>id: </span>{{hero.id}}</div>

<div>

<label>name:

<input [(ngModel)]="hero.name" placeholder="name"/>

</label>

</div>

</div>

Add the @Input() hero property

The HeroDetailComponent template binds to the component's hero property which is of type Hero.

Open the HeroDetailComponent class file and import the Hero symbol.

src/app/hero-detail/hero-detail.component.ts (import Hero)

content\_copy

import { Hero } from '../hero';

The hero property must be an Input property, annotated with the @Input() decorator, because the external HeroesComponent will bind to it like this.

content\_copy

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

Amend the @angular/core import statement to include the Input symbol.

src/app/hero-detail/hero-detail.component.ts (import Input)

content\_copy

import { Component, OnInit, Input } from '@angular/core';

Add a hero property, preceded by the @Input() decorator.

content\_copy

@Input() hero: Hero;

That's the only change you should make to the HeroDetailComponent class. There are no more properties. There's no presentation logic. This component simply receives a hero object through its hero property and displays it.

Show the HeroDetailComponent

The HeroesComponent is still a master/detail view.

It used to display the hero details on its own, before you cut that portion of the template. Now it will delegate to the HeroDetailComponent.

The two components will have a parent/child relationship. The parent HeroesComponent will control the child HeroDetailComponent by sending it a new hero to display whenever the user selects a hero from the list.

You won't change the HeroesComponent class but you will change its template.

Update the HeroesComponent template

The HeroDetailComponent selector is 'app-hero-detail'. Add an <app-hero-detail> element near the bottom of the HeroesComponent template, where the hero detail view used to be.

Bind the HeroesComponent.selectedHero to the element's hero property like this.

heroes.component.html (HeroDetail binding)

content\_copy

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

[hero]="selectedHero" is an Angular property binding.

It's a one way data binding from the selectedHero property of the HeroesComponent to the hero property of the target element, which maps to the hero property of the HeroDetailComponent.

Now when the user clicks a hero in the list, the selectedHero changes. When the selectedHero changes, the property binding updates hero and the HeroDetailComponent displays the new hero.

The revised HeroesComponent template should look like this:

heroes.component.html

content\_copy

<h2>My Heroes</h2>

<ul class="heroes">

<li \*ngFor="let hero of heroes"

[class.selected]="hero === selectedHero"

(click)="onSelect(hero)">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

</ul>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

The browser refreshes and the app starts working again as it did before.

What changed?

As before, whenever a user clicks on a hero name, the hero detail appears below the hero list. Now the HeroDetailComponent is presenting those details instead of the HeroesComponent.

Refactoring the original HeroesComponent into two components yields benefits, both now and in the future:

You simplified the HeroesComponent by reducing its responsibilities.

You can evolve the HeroDetailComponent into a rich hero editor without touching the parent HeroesComponent.

You can evolve the HeroesComponent without touching the hero detail view.

You can re-use the HeroDetailComponent in the template of some future component.

Final code review

Here are the code files discussed on this page and your app should look like this live example / download example.

src/app/hero-detail/hero-detail.component.ts

src/app/hero-detail/hero-detail.component.html

src/app/heroes/heroes.component.html

src/app/app.module.ts

content\_copy

import { Component, OnInit, Input } from '@angular/core';

import { Hero } from '../hero';

@Component({

selector: 'app-hero-detail',

templateUrl: './hero-detail.component.html',

styleUrls: ['./hero-detail.component.css']

})

export class HeroDetailComponent implements OnInit {

@Input() hero: Hero;

constructor() { }

ngOnInit() {

}

}

Summary

You created a separate, reusable HeroDetailComponent.

You used a property binding to give the parent HeroesComponent control over the child HeroDetailComponent.

You used the @Input decorator to make the hero property available for binding by the external HeroesComponent.

Services

The Tour of Heroes HeroesComponent is currently getting and displaying fake data.

After the refactoring in this tutorial, HeroesComponent will be lean and focused on supporting the view. It will also be easier to unit-test with a mock service.

Why services

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service.

In this tutorial, you'll create a HeroService that all application classes can use to get heroes. Instead of creating that service with new, you'll rely on Angular dependency injection to inject it into the HeroesComponent constructor.

Services are a great way to share information among classes that don't know each other. You'll create a MessageService and inject it in two places:

in HeroService which uses the service to send a message.

in MessagesComponent which displays that message.

Create the HeroService

Using the Angular CLI, create a service called hero.

content\_copy

ng generate service hero

The command generates skeleton HeroService class in src/app/hero.service.ts The HeroService class should look like the following example.

src/app/hero.service.ts (new service)

content\_copy

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root',

})

export class HeroService {

constructor() { }

}

@Injectable() services

Notice that the new service imports the Angular Injectable symbol and annotates the class with the @Injectable() decorator. This marks the class as one that participates in the dependency injection system. The HeroService class is going to provide an injectable service, and it can also have its own injected dependencies. It doesn't have any dependencies yet, but it will soon.

The @Injectable() decorator accepts a metadata object for the service, the same way the @Component() decorator did for your component classes.

Get hero data

The HeroService could get hero data from anywhere—a web service, local storage, or a mock data source.

Removing data access from components means you can change your mind about the implementation anytime, without touching any components. They don't know how the service works.

The implementation in this tutorial will continue to deliver mock heroes.

Import the Hero and HEROES.

content\_copy

import { Hero } from './hero';

import { HEROES } from './mock-heroes';

Add a getHeroes method to return the mock heroes.

content\_copy

getHeroes(): Hero[] {

return HEROES;

}

Provide the HeroService

You must make the HeroService available to the dependency injection system before Angular can inject it into the HeroesComponent, as you will do below. You do this by registering a provider. A provider is something that can create or deliver a service; in this case, it instantiates the HeroService class to provide the service.

Now, you need to make sure that the HeroService is registered as the provider of this service. You are registering it with an injector, which is the object that is responsible for choosing and injecting the provider where it is required.

By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the @Injectable decorator.

If you look at the @Injectable() statement right before the HeroService class definition, you can see that the providedIn metadata value is 'root':

content\_copy

@Injectable({

providedIn: 'root',

})

When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

To learn more about providers, see the Providers section. To learn more about injectors, see the Dependency Injection guide.

The HeroService is now ready to plug into the HeroesComponent.

This is a interim code sample that will allow you to provide and use the HeroService. At this point, the code will differ from the HeroService in the "final code review".

Update HeroesComponent

Open the HeroesComponent class file.

Delete the HEROES import, because you won't need that anymore. Import the HeroService instead.

src/app/heroes/heroes.component.ts (import HeroService)

content\_copy

import { HeroService } from '../hero.service';

Replace the definition of the heroes property with a simple declaration.

content\_copy

heroes: Hero[];

Inject the HeroService

Add a private heroService parameter of type HeroService to the constructor.

content\_copy

constructor(private heroService: HeroService) { }

The parameter simultaneously defines a private heroService property and identifies it as a HeroService injection site.

When Angular creates a HeroesComponent, the Dependency Injection system sets the heroService parameter to the singleton instance of HeroService.

Add getHeroes()

Create a function to retrieve the heroes from the service.

content\_copy

getHeroes(): void {

this.heroes = this.heroService.getHeroes();

}

Call it in ngOnInit

While you could call getHeroes() in the constructor, that's not the best practice.

Reserve the constructor for simple initialization such as wiring constructor parameters to properties. The constructor shouldn't do anything. It certainly shouldn't call a function that makes HTTP requests to a remote server as a real data service would.

Instead, call getHeroes() inside the ngOnInit lifecycle hook and let Angular call ngOnInit at an appropriate time after constructing a HeroesComponent instance.

content\_copy

ngOnInit() {

this.getHeroes();

}

See it run

After the browser refreshes, the app should run as before, showing a list of heroes and a hero detail view when you click on a hero name.

Observable data

The HeroService.getHeroes() method has a synchronous signature, which implies that the HeroService can fetch heroes synchronously. The HeroesComponent consumes the getHeroes() result as if heroes could be fetched synchronously.

content\_copy

this.heroes = this.heroService.getHeroes();

This will not work in a real app. You're getting away with it now because the service currently returns mock heroes. But soon the app will fetch heroes from a remote server, which is an inherently asynchronous operation.

The HeroService must wait for the server to respond, getHeroes() cannot return immediately with hero data, and the browser will not block while the service waits.

HeroService.getHeroes() must have an asynchronous signature of some kind.

It can take a callback. It could return a Promise. It could return an Observable.

In this tutorial, HeroService.getHeroes() will return an Observable in part because it will eventually use the Angular HttpClient.get method to fetch the heroes and HttpClient.get() returns an Observable.

Observable HeroService

Observable is one of the key classes in the RxJS library.

In a later tutorial on HTTP, you'll learn that Angular's HttpClient methods return RxJS Observables. In this tutorial, you'll simulate getting data from the server with the RxJS of() function.

Open the HeroService file and import the Observable and of symbols from RxJS.

src/app/hero.service.ts (Observable imports)

content\_copy

import { Observable, of } from 'rxjs';

Replace the getHeroes method with this one.

content\_copy

getHeroes(): Observable<Hero[]> {

return of(HEROES);

}

of(HEROES) returns an Observable<Hero[]> that emits a single value, the array of mock heroes.

In the HTTP tutorial, you'll call HttpClient.get<Hero[]>() which also returns an Observable<Hero[]> that emits a single value, an array of heroes from the body of the HTTP response.

Subscribe in HeroesComponent

The HeroService.getHeroes method used to return a Hero[]. Now it returns an Observable<Hero[]>.

You'll have to adjust to that difference in HeroesComponent.

Find the getHeroes method and replace it with the following code (shown side-by-side with the previous version for comparison)

heroes.component.ts (Observable)

heroes.component.ts (Original)

content\_copy

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes);

}

Observable.subscribe() is the critical difference.

The previous version assigns an array of heroes to the component's heroes property. The assignment occurs synchronously, as if the server could return heroes instantly or the browser could freeze the UI while it waited for the server's response.

That won't work when the HeroService is actually making requests of a remote server.

The new version waits for the Observable to emit the array of heroes— which could happen now or several minutes from now. Then subscribe passes the emitted array to the callback, which sets the component's heroes property.

This asynchronous approach will work when the HeroService requests heroes from the server.

Show messages

In this section you will

add a MessagesComponent that displays app messages at the bottom of the screen.

create an injectable, app-wide MessageService for sending messages to be displayed

inject MessageService into the HeroService

display a message when HeroService fetches heroes successfully.

Create MessagesComponent

Use the CLI to create the MessagesComponent.

content\_copy

ng generate component messages

The CLI creates the component files in the src/app/messages folder and declare MessagesComponent in AppModule.

Modify the AppComponent template to display the generated MessagesComponent

/src/app/app.component.html

content\_copy

<h1>{{title}}</h1>

<app-heroes></app-heroes>

<app-messages></app-messages>

You should see the default paragraph from MessagesComponent at the bottom of the page.

Create the MessageService

Use the CLI to create the MessageService in src/app.

content\_copy

ng generate service message

Open MessageService and replace its contents with the following.

/src/app/message.service.ts

content\_copy

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root',

})

export class MessageService {

messages: string[] = [];

add(message: string) {

this.messages.push(message);

}

clear() {

this.messages = [];

}

}

The service exposes its cache of messages and two methods: one to add() a message to the cache and another to clear() the cache.

Inject it into the HeroService

Re-open the HeroService and import the MessageService.

/src/app/hero.service.ts (import MessageService)

content\_copy

import { MessageService } from './message.service';

Modify the constructor with a parameter that declares a private messageService property. Angular will inject the singleton MessageService into that property when it creates the HeroService.

content\_copy

constructor(private messageService: MessageService) { }

This is a typical "service-in-service" scenario: you inject the MessageService into the HeroService which is injected into the HeroesComponent.

Send a message from HeroService

Modify the getHeroes method to send a message when the heroes are fetched.

content\_copy

getHeroes(): Observable<Hero[]> {

// TODO: send the message \_after\_ fetching the heroes

this.messageService.add('HeroService: fetched heroes');

return of(HEROES);

}

Display the message from HeroService

The MessagesComponent should display all messages, including the message sent by the HeroService when it fetches heroes.

Open MessagesComponent and import the MessageService.

/src/app/messages/messages.component.ts (import MessageService)

content\_copy

import { MessageService } from '../message.service';

Modify the constructor with a parameter that declares a public messageService property. Angular will inject the singleton MessageService into that property when it creates the MessagesComponent.

content\_copy

constructor(public messageService: MessageService) {}

The messageService property must be public because you're about to bind to it in the template.

Angular only binds to public component properties.

Bind to the MessageService

Replace the CLI-generated MessagesComponent template with the following.

src/app/messages/messages.component.html

content\_copy

<div \*ngIf="messageService.messages.length">

<h2>Messages</h2>

<button class="clear"

(click)="messageService.clear()">clear</button>

<div \*ngFor='let message of messageService.messages'> {{message}} </div>

</div>

This template binds directly to the component's messageService.

The \*ngIf only displays the messages area if there are messages to show.

An \*ngFor presents the list of messages in repeated <div> elements.

An Angular event binding binds the button's click event to MessageService.clear().

The messages will look better when you add the private CSS styles to messages.component.css as listed in one of the "final code review" tabs below.

The browser refreshes and the page displays the list of heroes. Scroll to the bottom to see the message from the HeroService in the message area. Click the "clear" button and the message area disappears.

Final code review

Here are the code files discussed on this page and your app should look like this live example / download example.

src/app/hero.service.ts

src/app/message.service.ts

src/app/heroes/heroes.component.ts

src/app/messages/messages.component.ts

src/app/messages/messages.component.html

src/app/messages/messages.component.css

src/app/app.module.ts

src/app/app.component.html

content\_copy

import { Injectable } from '@angular/core';

import { Observable, of } from 'rxjs';

import { Hero } from './hero';

import { HEROES } from './mock-heroes';

import { MessageService } from './message.service';

@Injectable({

providedIn: 'root',

})

export class HeroService {

constructor(private messageService: MessageService) { }

getHeroes(): Observable<Hero[]> {

// TODO: send the message \_after\_ fetching the heroes

this.messageService.add('HeroService: fetched heroes');

return of(HEROES);

}

}

Summary

You refactored data access to the HeroService class.

You registered the HeroService as the provider of its service at the root level so that it can be injected anywhere in the app.

You used Angular Dependency Injection to inject it into a component.

You gave the HeroService get data method an asynchronous signature.

You discovered Observable and the RxJS Observable library.

You used RxJS of() to return an observable of mock heroes (Observable<Hero[]>).

The component's ngOnInit lifecycle hook calls the HeroService method, not the constructor.

You created a MessageService for loosely-coupled communication between classes.

The HeroService injected into a component is created with another injected service, MessageService.

Routing

There are new requirements for the Tour of Heroes app:

Add a Dashboard view.

Add the ability to navigate between the Heroes and Dashboard views.

When users click a hero name in either view, navigate to a detail view of the selected hero.

When users click a deep link in an email, open the detail view for a particular hero.

When you’re done, users will be able to navigate the app like this:

View navigations

Add the AppRoutingModule

An Angular best practice is to load and configure the router in a separate, top-level module that is dedicated to routing and imported by the root AppModule.

By convention, the module class name is AppRoutingModule and it belongs in the app-routing.module.ts in the src/app folder.

Use the CLI to generate it.

content\_copy

ng generate module app-routing --flat --module=app

--flat puts the file in src/app instead of its own folder.

--module=app tells the CLI to register it in the imports array of the AppModule.

The generated file looks like this:

src/app/app-routing.module.ts (generated)

content\_copy

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

@NgModule({

imports: [

CommonModule

],

declarations: []

})

export class AppRoutingModule { }

You generally don't declare components in a routing module so you can delete the @NgModule.declarations array and delete CommonModule references too.

You'll configure the router with Routes in the RouterModule so import those two symbols from the @angular/router library.

Add an @NgModule.exports array with RouterModule in it. Exporting RouterModule makes router directives available for use in the AppModule components that will need them.

AppRoutingModule looks like this now:

src/app/app-routing.module.ts (v1)

content\_copy

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

@NgModule({

exports: [ RouterModule ]

})

export class AppRoutingModule {}

Add routes

Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar.

A typical Angular Route has two properties:

path: a string that matches the URL in the browser address bar.

component: the component that the router should create when navigating to this route.

You intend to navigate to the HeroesComponent when the URL is something like localhost:4200/heroes.

Import the HeroesComponent so you can reference it in a Route. Then define an array of routes with a single route to that component.

content\_copy

import { HeroesComponent } from './heroes/heroes.component';

const routes: Routes = [

{ path: 'heroes', component: HeroesComponent }

];

Once you've finished setting up, the router will match that URL to path: 'heroes' and display the HeroesComponent.

RouterModule.forRoot()

You first must initialize the router and start it listening for browser location changes.

Add RouterModule to the @NgModule.imports array and configure it with the routes in one step by calling RouterModule.forRoot() within the imports array, like this:

content\_copy

imports: [ RouterModule.forRoot(routes) ],

The method is called forRoot() because you configure the router at the application's root level. The forRoot() method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

Add RouterOutlet

Open the AppComponent template replace the <app-heroes> element with a <router-outlet> element.

src/app/app.component.html (router-outlet)

content\_copy

<h1>{{title}}</h1>

<router-outlet></router-outlet>

<app-messages></app-messages>

You removed <app-heroes> because you will only display the HeroesComponent when the user navigates to it.

The <router-outlet> tells the router where to display routed views.

The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported RouterModule.

Try it

You should still be running with this CLI command.

content\_copy

ng serve

The browser should refresh and display the app title but not the list of heroes.

Look at the browser's address bar. The URL ends in /. The route path to HeroesComponent is /heroes.

Append /heroes to the URL in the browser address bar. You should see the familiar heroes master/detail view.

Add a navigation link (routerLink)

Users shouldn't have to paste a route URL into the address bar. They should be able to click a link to navigate.

Add a <nav> element and, within that, an anchor element that, when clicked, triggers navigation to the HeroesComponent. The revised AppComponent template looks like this:

src/app/app.component.html (heroes RouterLink)

content\_copy

<h1>{{title}}</h1>

<nav>

<a routerLink="/heroes">Heroes</a>

</nav>

<router-outlet></router-outlet>

<app-messages></app-messages>

A routerLink attribute is set to "/heroes", the string that the router matches to the route to HeroesComponent. The routerLink is the selector for the RouterLink directive that turns user clicks into router navigations. It's another of the public directives in the RouterModule.

The browser refreshes and displays the app title and heroes link, but not the heroes list.

Click the link. The address bar updates to /heroes and the list of heroes appears.

Make this and future navigation links look better by adding private CSS styles to app.component.css as listed in the final code review below.

Add a dashboard view

Routing makes more sense when there are multiple views. So far there's only the heroes view.

Add a DashboardComponent using the CLI:

content\_copy

ng generate component dashboard

The CLI generates the files for the DashboardComponent and declares it in AppModule.

Replace the default file content in these three files as follows and then return for a little discussion:

src/app/dashboard/dashboard.component.html

src/app/dashboard/dashboard.component.ts

src/app/dashboard/dashboard.component.css

content\_copy

<h3>Top Heroes</h3>

<div class="grid grid-pad">

<a \*ngFor="let hero of heroes" class="col-1-4">

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

</a>

</div>

The template presents a grid of hero name links.

The \*ngFor repeater creates as many links as are in the component's heroes array.

The links are styled as colored blocks by the dashboard.component.css.

The links don't go anywhere yet but they will shortly.

The class is similar to the HeroesComponent class.

It defines a heroes array property.

The constructor expects Angular to inject the HeroService into a private heroService property.

The ngOnInit() lifecycle hook calls getHeroes.

This getHeroes reduces the number of heroes displayed to four (2nd, 3rd, 4th, and 5th).

content\_copy

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes.slice(1, 5));

}

Add the dashboard route

To navigate to the dashboard, the router needs an appropriate route.

Import the DashboardComponent in the AppRoutingModule.

src/app/app-routing.module.ts (import DashboardComponent)

content\_copy

import { DashboardComponent } from './dashboard/dashboard.component';

Add a route to the AppRoutingModule.routes array that matches a path to the DashboardComponent.

content\_copy

{ path: 'dashboard', component: DashboardComponent },

Add a default route

When the app starts, the browsers address bar points to the web site's root. That doesn't match any existing route so the router doesn't navigate anywhere. The space below the <router-outlet> is blank.

To make the app navigate to the dashboard automatically, add the following route to the AppRoutingModule.Routes array.

content\_copy

{ path: '', redirectTo: '/dashboard', pathMatch: 'full' },

This route redirects a URL that fully matches the empty path to the route whose path is '/dashboard'.

After the browser refreshes, the router loads the DashboardComponent and the browser address bar shows the /dashboard URL.

Add dashboard link to the shell

The user should be able to navigate back and forth between the DashboardComponent and the HeroesComponent by clicking links in the navigation area near the top of the page.

Add a dashboard navigation link to the AppComponent shell template, just above the Heroes link.

src/app/app.component.html

content\_copy

<h1>{{title}}</h1>

<nav>

<a routerLink="/dashboard">Dashboard</a>

<a routerLink="/heroes">Heroes</a>

</nav>

<router-outlet></router-outlet>

<app-messages></app-messages>

After the browser refreshes you can navigate freely between the two views by clicking the links.

Navigating to hero details

The HeroDetailsComponent displays details of a selected hero. At the moment the HeroDetailsComponent is only visible at the bottom of the HeroesComponent

The user should be able to get to these details in three ways.

By clicking a hero in the dashboard.

By clicking a hero in the heroes list.

By pasting a "deep link" URL into the browser address bar that identifies the hero to display.

In this section, you'll enable navigation to the HeroDetailsComponent and liberate it from the HeroesComponent.

Delete hero details from HeroesComponent

When the user clicks a hero item in the HeroesComponent, the app should navigate to the HeroDetailComponent, replacing the heroes list view with the hero detail view. The heroes list view should no longer show hero details as it does now.

Open the HeroesComponent template (heroes/heroes.component.html) and delete the <app-hero-detail> element from the bottom.

Clicking a hero item now does nothing. You'll fix that shortly after you enable routing to the HeroDetailComponent.

Add a hero detail route

A URL like ~/detail/11 would be a good URL for navigating to the Hero Detail view of the hero whose id is 11.

Open AppRoutingModule and import HeroDetailComponent.

src/app/app-routing.module.ts (import HeroDetailComponent)

content\_copy

import { HeroDetailComponent } from './hero-detail/hero-detail.component';

Then add a parameterized route to the AppRoutingModule.routes array that matches the path pattern to the hero detail view.

content\_copy

{ path: 'detail/:id', component: HeroDetailComponent },

The colon (:) in the path indicates that :id is a placeholder for a specific hero id.

At this point, all application routes are in place.

src/app/app-routing.module.ts (all routes)

content\_copy

const routes: Routes = [

{ path: '', redirectTo: '/dashboard', pathMatch: 'full' },

{ path: 'dashboard', component: DashboardComponent },

{ path: 'detail/:id', component: HeroDetailComponent },

{ path: 'heroes', component: HeroesComponent }

];

DashboardComponent hero links

The DashboardComponent hero links do nothing at the moment.

Now that the router has a route to HeroDetailComponent, fix the dashboard hero links to navigate via the parameterized dashboard route.

src/app/dashboard/dashboard.component.html (hero links)

content\_copy

<a \*ngFor="let hero of heroes" class="col-1-4"

routerLink="/detail/{{hero.id}}">

You're using Angular interpolation binding within the \*ngFor repeater to insert the current iteration's hero.id into each routerLink.

HeroesComponent hero links

The hero items in the HeroesComponent are <li> elements whose click events are bound to the component's onSelect() method.

src/app/heroes/heroes.component.html (list with onSelect)

content\_copy

<ul class="heroes">

<li \*ngFor="let hero of heroes"

[class.selected]="hero === selectedHero"

(click)="onSelect(hero)">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

</ul>

Strip the <li> back to just its \*ngFor, wrap the badge and name in an anchor element (<a>), and add a routerLink attribute to the anchor that is the same as in the dashboard template

src/app/heroes/heroes.component.html (list with links)

content\_copy

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<a routerLink="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</a>

</li>

</ul>

You'll have to fix the private stylesheet (heroes.component.css) to make the list look as it did before. Revised styles are in the final code review at the bottom of this guide.

Remove dead code (optional)

While the HeroesComponent class still works, the onSelect() method and selectedHero property are no longer used.

It's nice to tidy up and you'll be grateful to yourself later. Here's the class after pruning away the dead code.

src/app/heroes/heroes.component.ts (cleaned up)

content\_copy

export class HeroesComponent implements OnInit {

heroes: Hero[];

constructor(private heroService: HeroService) { }

ngOnInit() {

this.getHeroes();

}

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes);

}

}

Routable HeroDetailComponent

Previously, the parent HeroesComponent set the HeroDetailComponent.hero property and the HeroDetailComponent displayed the hero.

HeroesComponent doesn't do that anymore. Now the router creates the HeroDetailComponent in response to a URL such as ~/detail/11.

The HeroDetailComponent needs a new way to obtain the hero-to-display.

Get the route that created it,

Extract the id from the route

Acquire the hero with that id from the server via the HeroService

Add the following imports:

src/app/hero-detail/hero-detail.component.ts

content\_copy

import { ActivatedRoute } from '@angular/router';

import { Location } from '@angular/common';

import { HeroService } from '../hero.service';

Inject the ActivatedRoute, HeroService, and Location services into the constructor, saving their values in private fields:

content\_copy

constructor(

private route: ActivatedRoute,

private heroService: HeroService,

private location: Location

) {}

The ActivatedRoute holds information about the route to this instance of the HeroDetailComponent. This component is interested in the route's bag of parameters extracted from the URL. The "id" parameter is the id of the hero to display.

The HeroService gets hero data from the remote server and this component will use it to get the hero-to-display.

The location is an Angular service for interacting with the browser. You'll use it later to navigate back to the view that navigated here.

Extract the id route parameter

In the ngOnInit() lifecycle hook call getHero() and define it as follows.

content\_copy

ngOnInit(): void {

this.getHero();

}

getHero(): void {

const id = +this.route.snapshot.paramMap.get('id');

this.heroService.getHero(id)

.subscribe(hero => this.hero = hero);

}

The route.snapshot is a static image of the route information shortly after the component was created.

The paramMap is a dictionary of route parameter values extracted from the URL. The "id" key returns the id of the hero to fetch.

Route parameters are always strings. The JavaScript (+) operator converts the string to a number, which is what a hero id should be.

The browser refreshes and the app crashes with a compiler error. HeroService doesn't have a getHero() method. Add it now.

Add HeroService.getHero()

Open HeroService and add this getHero() method

src/app/hero.service.ts (getHero)

content\_copy

getHero(id: number): Observable<Hero> {

// TODO: send the message \_after\_ fetching the hero

this.messageService.add(`HeroService: fetched hero id=${id}`);

return of(HEROES.find(hero => hero.id === id));

}

Note the backticks ( ` ) that define a JavaScript template literal for embedding the id.

Like getHeroes(), getHero() has an asynchronous signature. It returns a mock hero as an Observable, using the RxJS of() function.

You'll be able to re-implement getHero() as a real Http request without having to change the HeroDetailComponent that calls it.

Try it

The browser refreshes and the app is working again. You can click a hero in the dashboard or in the heroes list and navigate to that hero's detail view.

If you paste localhost:4200/detail/11 in the browser address bar, the router navigates to the detail view for the hero with id: 11, "Mr. Nice".

Find the way back

By clicking the browser's back button, you can go back to the hero list or dashboard view, depending upon which sent you to the detail view.

It would be nice to have a button on the HeroDetail view that can do that.

Add a go back button to the bottom of the component template and bind it to the component's goBack() method.

src/app/hero-detail/hero-detail.component.html (back button)

content\_copy

<button (click)="goBack()">go back</button>

Add a goBack() method to the component class that navigates backward one step in the browser's history stack using the Location service that you injected previously.

src/app/hero-detail/hero-detail.component.ts (goBack)

content\_copy

goBack(): void {

this.location.back();

}

Refresh the browser and start clicking. Users can navigate around the app, from the dashboard to hero details and back, from heroes list to the mini detail to the hero details and back to the heroes again.

You've met all of the navigational requirements that propelled this page.

Final code review

Here are the code files discussed on this page and your app should look like this live example / download example.

AppRoutingModule, AppModule, and HeroService

src/app/app-routing.module.ts

src/app/app.module.ts

src/app/hero.service.ts

content\_copy

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { DashboardComponent } from './dashboard/dashboard.component';

import { HeroesComponent } from './heroes/heroes.component';

import { HeroDetailComponent } from './hero-detail/hero-detail.component';

const routes: Routes = [

{ path: '', redirectTo: '/dashboard', pathMatch: 'full' },

{ path: 'dashboard', component: DashboardComponent },

{ path: 'detail/:id', component: HeroDetailComponent },

{ path: 'heroes', component: HeroesComponent }

];

@NgModule({

imports: [ RouterModule.forRoot(routes) ],

exports: [ RouterModule ]

})

export class AppRoutingModule {}

AppComponent

src/app/app.component.html

src/app/app.component.css

content\_copy

<h1>{{title}}</h1>

<nav>

<a routerLink="/dashboard">Dashboard</a>

<a routerLink="/heroes">Heroes</a>

</nav>

<router-outlet></router-outlet>

<app-messages></app-messages>

DashboardComponent

src/app/dashboard/dashboard.component.html

src/app/dashboard/dashboard.component.ts

src/app/dashboard/dashboard.component.css

content\_copy

<h3>Top Heroes</h3>

<div class="grid grid-pad">

<a \*ngFor="let hero of heroes" class="col-1-4"

routerLink="/detail/{{hero.id}}">

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

</a>

</div>

HeroesComponent

src/app/heroes/heroes.component.html

src/app/heroes/heroes.component.ts

src/app/heroes/heroes.component.css

content\_copy

<h2>My Heroes</h2>

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<a routerLink="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</a>

</li>

</ul>

HeroDetailComponent

src/app/hero-detail/hero-detail.component.html

src/app/hero-detail/hero-detail.component.ts

src/app/hero-detail/hero-detail.component.css

content\_copy

<div \*ngIf="hero">

<h2>{{hero.name | uppercase}} Details</h2>

<div><span>id: </span>{{hero.id}}</div>

<div>

<label>name:

<input [(ngModel)]="hero.name" placeholder="name"/>

</label>

</div>

<button (click)="goBack()">go back</button>

</div>

Summary

You added the Angular router to navigate among different components.

You turned the AppComponent into a navigation shell with <a> links and a <router-outlet>.

You configured the router in an AppRoutingModule

You defined simple routes, a redirect route, and a parameterized route.

You used the routerLink directive in anchor elements.

You refactored a tightly-coupled master/detail view into a routed detail view.

You used router link parameters to navigate to the detail view of a user-selected hero.

You shared the HeroService among multiple components.

HTTP

In this tutorial, you'll add the following data persistence features with help from Angular's HttpClient.

The HeroService gets hero data with HTTP requests.

Users can add, edit, and delete heroes and save these changes over HTTP.

Users can search for heroes by name.

When you're done with this page, the app should look like this live example / download example.

Enable HTTP services

HttpClient is Angular's mechanism for communicating with a remote server over HTTP.

To make HttpClient available everywhere in the app,

open the root AppModule,

import the HttpClientModule symbol from @angular/common/http,

add it to the @NgModule.imports array.

Simulate a data server

This tutorial sample mimics communication with a remote data server by using the In-memory Web API module.

After installing the module, the app will make requests to and receive responses from the HttpClient without knowing that the In-memory Web API is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

This facility is a great convenience for the tutorial. You won't have to set up a server to learn about HttpClient.

It may also be convenient in the early stages of your own app development when the server's web api is ill-defined or not yet implemented.

Important: the In-memory Web API module has nothing to do with HTTP in Angular.

If you're just reading this tutorial to learn about HttpClient, you can skip over this step. If you're coding along with this tutorial, stay here and add the In-memory Web API now.

Install the In-memory Web API package from npm

content\_copy

npm install angular-in-memory-web-api --save

Import the HttpClientInMemoryWebApiModule and the InMemoryDataService class, which you will create in a moment.

src/app/app.module.ts (In-memory Web API imports)

content\_copy

import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';

import { InMemoryDataService } from './in-memory-data.service';

Add the HttpClientInMemoryWebApiModule to the @NgModule.imports array— after importing the HttpClient, —while configuring it with the InMemoryDataService.

content\_copy

HttpClientModule,

// The HttpClientInMemoryWebApiModule module intercepts HTTP requests

// and returns simulated server responses.

// Remove it when a real server is ready to receive requests.

HttpClientInMemoryWebApiModule.forRoot(

InMemoryDataService, { dataEncapsulation: false }

)

The forRoot() configuration method takes an InMemoryDataService class that primes the in-memory database.

The Tour of Heroes sample creates such a class src/app/in-memory-data.service.ts which has the following content:

src/app/in-memory-data.service.ts

content\_copy

import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemoryDataService implements InMemoryDbService {

createDb() {

const heroes = [

{ id: 11, name: 'Mr. Nice' },

{ id: 12, name: 'Narco' },

{ id: 13, name: 'Bombasto' },

{ id: 14, name: 'Celeritas' },

{ id: 15, name: 'Magneta' },

{ id: 16, name: 'RubberMan' },

{ id: 17, name: 'Dynama' },

{ id: 18, name: 'Dr IQ' },

{ id: 19, name: 'Magma' },

{ id: 20, name: 'Tornado' }

];

return {heroes};

}

}

This file replaces mock-heroes.ts, which is now safe to delete.

When your server is ready, detach the In-memory Web API, and the app's requests will go through to the server.

Now back to the HttpClient story.

Heroes and HTTP

Import some HTTP symbols that you'll need:

src/app/hero.service.ts (import HTTP symbols)

content\_copy

import { HttpClient, HttpHeaders } from '@angular/common/http';

Inject HttpClient into the constructor in a private property called http.

content\_copy

constructor(

private http: HttpClient,

private messageService: MessageService) { }

Keep injecting the MessageService. You'll call it so frequently that you'll wrap it in private log method.

content\_copy

/\*\* Log a HeroService message with the MessageService \*/

private log(message: string) {

this.messageService.add(`HeroService: ${message}`);

}

Define the heroesUrl with the address of the heroes resource on the server.

content\_copy

private heroesUrl = 'api/heroes'; // URL to web api

Get heroes with HttpClient

The current HeroService.getHeroes() uses the RxJS of() function to return an array of mock heroes as an Observable<Hero[]>.

src/app/hero.service.ts (getHeroes with RxJs 'of()')

content\_copy

getHeroes(): Observable<Hero[]> {

return of(HEROES);

}

Convert that method to use HttpClient

content\_copy

/\*\* GET heroes from the server \*/

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

}

Refresh the browser. The hero data should successfully load from the mock server.

You've swapped of for http.get and the app keeps working without any other changes because both functions return an Observable<Hero[]>.

Http methods return one value

All HttpClient methods return an RxJS Observable of something.

HTTP is a request/response protocol. You make a request, it returns a single response.

In general, an observable can return multiple values over time. An observable from HttpClient always emits a single value and then completes, never to emit again.

This particular HttpClient.get call returns an Observable<Hero[]>, literally "an observable of hero arrays". In practice, it will only return a single hero array.

HttpClient.get returns response data

HttpClient.get returns the body of the response as an untyped JSON object by default. Applying the optional type specifier, <Hero[]> , gives you a typed result object.

The shape of the JSON data is determined by the server's data API. The Tour of Heroes data API returns the hero data as an array.

Other APIs may bury the data that you want within an object. You might have to dig that data out by processing the Observable result with the RxJS map operator.

Although not discussed here, there's an example of map in the getHeroNo404() method included in the sample source code.

Error handling

Things go wrong, especially when you're getting data from a remote server. The HeroService.getHeroes() method should catch errors and do something appropriate.

To catch errors, you "pipe" the observable result from http.get() through an RxJS catchError() operator.

Import the catchError symbol from rxjs/operators, along with some other operators you'll need later.

content\_copy

import { catchError, map, tap } from 'rxjs/operators';

Now extend the observable result with the .pipe() method and give it a catchError() operator.

content\_copy

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

.pipe(

catchError(this.handleError('getHeroes', []))

);

}

The catchError() operator intercepts an Observable that failed. It passes the error an error handler that can do what it wants with the error.

The following handleError() method reports the error and then returns an innocuous result so that the application keeps working.

handleError

The following errorHandler() will be shared by many HeroService methods so it's generalized to meet their different needs.

Instead of handling the error directly, it returns an error handler function to catchError that it has configured with both the name of the operation that failed and a safe return value.

content\_copy

/\*\*

\* Handle Http operation that failed.

\* Let the app continue.

\* @param operation - name of the operation that failed

\* @param result - optional value to return as the observable result

\*/

private handleError<T> (operation = 'operation', result?: T) {

return (error: any): Observable<T> => {

// TODO: send the error to remote logging infrastructure

console.error(error); // log to console instead

// TODO: better job of transforming error for user consumption

this.log(`${operation} failed: ${error.message}`);

// Let the app keep running by returning an empty result.

return of(result as T);

};

}

After reporting the error to console, the handler constructs a user friendly message and returns a safe value to the app so it can keep working.

Because each service method returns a different kind of Observable result, errorHandler() takes a type parameter so it can return the safe value as the type that the app expects.

Tap into the Observable

The HeroService methods will tap into the flow of observable values and send a message (via log()) to the message area at the bottom of the page.

They'll do that with the RxJS tap operator, which looks at the observable values, does something with those values, and passes them along. The tap call back doesn't touch the values themselves.

Here is the final version of getHeroes with the tap that logs the operation.

content\_copy

/\*\* GET heroes from the server \*/

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

.pipe(

tap(heroes => this.log('fetched heroes')),

catchError(this.handleError('getHeroes', []))

);

}

Get hero by id

Most web APIs support a get by id request in the form api/hero/:id (such as api/hero/11). Add a HeroService.getHero() method to make that request:

src/app/hero.service.ts

content\_copy

/\*\* GET hero by id. Will 404 if id not found \*/

getHero(id: number): Observable<Hero> {

const url = `${this.heroesUrl}/${id}`;

return this.http.get<Hero>(url).pipe(

tap(\_ => this.log(`fetched hero id=${id}`)),

catchError(this.handleError<Hero>(`getHero id=${id}`))

);

}

There are three significant differences from getHeroes().

it constructs a request URL with the desired hero's id.

the server should respond with a single hero rather than an array of heroes.

therefore, getHero returns an Observable<Hero> ("an observable of Hero objects") rather than an observable of hero arrays .

Update heroes

Edit a hero's name in the hero detail view. As you type, the hero name updates the heading at the top of the page. But when you click the "go back button", the changes are lost.

If you want changes to persist, you must write them back to the server.

At the end of the hero detail template, add a save button with a click event binding that invokes a new component method named save().

src/app/hero-detail/hero-detail.component.html (save)

content\_copy

<button (click)="save()">save</button>

Add the following save() method, which persists hero name changes using the hero service updateHero() method and then navigates back to the previous view.

src/app/hero-detail/hero-detail.component.ts (save)

content\_copy

save(): void {

this.heroService.updateHero(this.hero)

.subscribe(() => this.goBack());

}

Add HeroService.updateHero()

The overall structure of the updateHero() method is similar to that of getHeroes(), but it uses http.put() to persist the changed hero on the server.

src/app/hero.service.ts (update)

content\_copy

/\*\* PUT: update the hero on the server \*/

updateHero (hero: Hero): Observable<any> {

return this.http.put(this.heroesUrl, hero, httpOptions).pipe(

tap(\_ => this.log(`updated hero id=${hero.id}`)),

catchError(this.handleError<any>('updateHero'))

);

}

The HttpClient.put() method takes three parameters

the URL

the data to update (the modified hero in this case)

options

The URL is unchanged. The heroes web API knows which hero to update by looking at the hero's id.

The heroes web API expects a special header in HTTP save requests. That header is in the httpOptions constant defined in the HeroService.

src/app/hero.service.ts

content\_copy

const httpOptions = {

headers: new HttpHeaders({ 'Content-Type': 'application/json' })

};

Refresh the browser, change a hero name and save your change. Navigating to the previous view is implemented in the save() method defined in HeroDetailComponent. The hero now appears in the list with the changed name.

Add a new hero

To add a hero, this app only needs the hero's name. You can use an input element paired with an add button.

Insert the following into the HeroesComponent template, just after the heading:

src/app/heroes/heroes.component.html (add)

content\_copy

<div>

<label>Hero name:

<input #heroName />

</label>

<!-- (click) passes input value to add() and then clears the input -->

<button (click)="add(heroName.value); heroName.value=''">

add

</button>

</div>

In response to a click event, call the component's click handler and then clear the input field so that it's ready for another name.

src/app/heroes/heroes.component.ts (add)

content\_copy

add(name: string): void {

name = name.trim();

if (!name) { return; }

this.heroService.addHero({ name } as Hero)

.subscribe(hero => {

this.heroes.push(hero);

});

}

When the given name is non-blank, the handler creates a Hero-like object from the name (it's only missing the id) and passes it to the services addHero() method.

When addHero saves successfully, the subscribe callback receives the new hero and pushes it into to the heroes list for display.

You'll write HeroService.addHero in the next section.

Add HeroService.addHero()

Add the following addHero() method to the HeroService class.

src/app/hero.service.ts (addHero)

content\_copy

/\*\* POST: add a new hero to the server \*/

addHero (hero: Hero): Observable<Hero> {

return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(

tap((hero: Hero) => this.log(`added hero w/ id=${hero.id}`)),

catchError(this.handleError<Hero>('addHero'))

);

}

HeroService.addHero() differs from updateHero in two ways.

it calls HttpClient.post() instead of put().

it expects the server to generates an id for the new hero, which it returns in the Observable<Hero> to the caller.

Refresh the browser and add some heroes.

Delete a hero

Each hero in the heroes list should have a delete button.

Add the following button element to the HeroesComponent template, after the hero name in the repeated <li> element.

content\_copy

<button class="delete" title="delete hero"

(click)="delete(hero)">x</button>

The HTML for the list of heroes should look like this:

src/app/heroes/heroes.component.html (list of heroes)

content\_copy

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<a routerLink="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</a>

<button class="delete" title="delete hero"

(click)="delete(hero)">x</button>

</li>

</ul>

To position the delete button at the far right of the hero entry, add some CSS to the heroes.component.css. You'll find that CSS in the final review code below.

Add the delete() handler to the component.

src/app/heroes/heroes.component.ts (delete)

content\_copy

delete(hero: Hero): void {

this.heroes = this.heroes.filter(h => h !== hero);

this.heroService.deleteHero(hero).subscribe();

}

Although the component delegates hero deletion to the HeroService, it remains responsible for updating its own list of heroes. The component's delete() method immediately removes the hero-to-delete from that list, anticipating that the HeroService will succeed on the server.

There's really nothing for the component to do with the Observable returned by heroService.delete(). It must subscribe anyway.

If you neglect to subscribe(), the service will not send the delete request to the server! As a rule, an Observable does nothing until something subscribes!

Confirm this for yourself by temporarily removing the subscribe(), clicking "Dashboard", then clicking "Heroes". You'll see the full list of heroes again.

Add HeroService.deleteHero()

Add a deleteHero() method to HeroService like this.

src/app/hero.service.ts (delete)

content\_copy

/\*\* DELETE: delete the hero from the server \*/

deleteHero (hero: Hero | number): Observable<Hero> {

const id = typeof hero === 'number' ? hero : hero.id;

const url = `${this.heroesUrl}/${id}`;

return this.http.delete<Hero>(url, httpOptions).pipe(

tap(\_ => this.log(`deleted hero id=${id}`)),

catchError(this.handleError<Hero>('deleteHero'))

);

}

Note that

it calls HttpClient.delete.

the URL is the heroes resource URL plus the id of the hero to delete

you don't send data as you did with put and post.

you still send the httpOptions.

Refresh the browser and try the new delete functionality.

Search by name

In this last exercise, you learn to chain Observable operators together so you can minimize the number of similar HTTP requests and consume network bandwidth economically.

You will add a heroes search feature to the Dashboard. As the user types a name into a search box, you'll make repeated HTTP requests for heroes filtered by that name. Your goal is to issue only as many requests as necessary.

HeroService.searchHeroes

Start by adding a searchHeroes method to the HeroService.

src/app/hero.service.ts

content\_copy

/\* GET heroes whose name contains search term \*/

searchHeroes(term: string): Observable<Hero[]> {

if (!term.trim()) {

// if not search term, return empty hero array.

return of([]);

}

return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(

tap(\_ => this.log(`found heroes matching "${term}"`)),

catchError(this.handleError<Hero[]>('searchHeroes', []))

);

}

The method returns immediately with an empty array if there is no search term. The rest of it closely resembles getHeroes(). The only significant difference is the URL, which includes a query string with the search term.

Add search to the Dashboard

Open the DashboardComponent template and Add the hero search element, <app-hero-search>, to the bottom of the DashboardComponent template.

src/app/dashboard/dashboard.component.html

content\_copy

<h3>Top Heroes</h3>

<div class="grid grid-pad">

<a \*ngFor="let hero of heroes" class="col-1-4"

routerLink="/detail/{{hero.id}}">

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

</a>

</div>

<app-hero-search></app-hero-search>

This template looks a lot like the \*ngFor repeater in the HeroesComponent template.

Unfortunately, adding this element breaks the app. Angular can't find a component with a selector that matches <app-hero-search>.

The HeroSearchComponent doesn't exist yet. Fix that.

Create HeroSearchComponent

Create a HeroSearchComponent with the CLI.

content\_copy

ng generate component hero-search

The CLI generates the three HeroSearchComponent files and adds the component to the AppModule declarations

Replace the generated HeroSearchComponent template with a text box and a list of matching search results like this.

src/app/hero-search/hero-search.component.html

content\_copy

<div id="search-component">

<h4>Hero Search</h4>

<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

<ul class="search-result">

<li \*ngFor="let hero of heroes$ | async" >

<a routerLink="/detail/{{hero.id}}">

{{hero.name}}

</a>

</li>

</ul>

</div>

Add private CSS styles to hero-search.component.css as listed in the final code review below.

As the user types in the search box, a keyup event binding calls the component's search() method with the new search box value.

AsyncPipe

As expected, the \*ngFor repeats hero objects.

Look closely and you'll see that the \*ngFor iterates over a list called heroes$, not heroes.

content\_copy

<li \*ngFor="let hero of heroes$ | async" >

The $ is a convention that indicates heroes$ is an Observable, not an array.

The \*ngFor can't do anything with an Observable. But there's also a pipe character (|) followed by async, which identifies Angular's AsyncPipe.

The AsyncPipe subscribes to an Observable automatically so you won't have to do so in the component class.

Fix the HeroSearchComponent class

Replace the generated HeroSearchComponent class and metadata as follows.

src/app/hero-search/hero-search.component.ts

content\_copy

import { Component, OnInit } from '@angular/core';

import { Observable, Subject } from 'rxjs';

import {

debounceTime, distinctUntilChanged, switchMap

} from 'rxjs/operators';

import { Hero } from '../hero';

import { HeroService } from '../hero.service';

@Component({

selector: 'app-hero-search',

templateUrl: './hero-search.component.html',

styleUrls: [ './hero-search.component.css' ]

})

export class HeroSearchComponent implements OnInit {

heroes$: Observable<Hero[]>;

private searchTerms = new Subject<string>();

constructor(private heroService: HeroService) {}

// Push a search term into the observable stream.

search(term: string): void {

this.searchTerms.next(term);

}

ngOnInit(): void {

this.heroes$ = this.searchTerms.pipe(

// wait 300ms after each keystroke before considering the term

debounceTime(300),

// ignore new term if same as previous term

distinctUntilChanged(),

// switch to new search observable each time the term changes

switchMap((term: string) => this.heroService.searchHeroes(term)),

);

}

}

Notice the declaration of heroes$ as an Observable

content\_copy

heroes$: Observable<Hero[]>;

You'll set it in ngOnInit(). Before you do, focus on the definition of searchTerms.

The searchTerms RxJS subject

The searchTerms property is declared as an RxJS Subject.

content\_copy

private searchTerms = new Subject<string>();

// Push a search term into the observable stream.

search(term: string): void {

this.searchTerms.next(term);

}

A Subject is both a source of observable values and an Observable itself. You can subscribe to a Subject as you would any Observable.

You can also push values into that Observable by calling its next(value) method as the search() method does.

The search() method is called via an event binding to the textbox's keystroke event.

content\_copy

<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

Every time the user types in the textbox, the binding calls search() with the textbox value, a "search term". The searchTerms becomes an Observable emitting a steady stream of search terms.

Chaining RxJS operators

Passing a new search term directly to the searchHeroes() after every user keystroke would create an excessive amount of HTTP requests, taxing server resources and burning through the cellular network data plan.

Instead, the ngOnInit() method pipes the searchTerms observable through a sequence of RxJS operators that reduce the number of calls to the searchHeroes(), ultimately returning an observable of timely hero search results (each a Hero[]).

Here's the code.

content\_copy

this.heroes$ = this.searchTerms.pipe(

// wait 300ms after each keystroke before considering the term

debounceTime(300),

// ignore new term if same as previous term

distinctUntilChanged(),

// switch to new search observable each time the term changes

switchMap((term: string) => this.heroService.searchHeroes(term)),

);

debounceTime(300) waits until the flow of new string events pauses for 300 milliseconds before passing along the latest string. You'll never make requests more frequently than 300ms.

distinctUntilChanged() ensures that a request is sent only if the filter text changed.

switchMap() calls the search service for each search term that makes it through debounce and distinctUntilChanged. It cancels and discards previous search observables, returning only the latest search service observable.

With the switchMap operator, every qualifying key event can trigger an HttpClient.get() method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.

switchMap() preserves the original request order while returning only the observable from the most recent HTTP method call. Results from prior calls are canceled and discarded.

Note that canceling a previous searchHeroes() Observable doesn't actually abort a pending HTTP request. Unwanted results are simply discarded before they reach your application code.

Remember that the component class does not subscribe to the heroes$ observable. That's the job of the AsyncPipe in the template.

Try it

Run the app again. In the Dashboard, enter some text in the search box. If you enter characters that match any existing hero names, you'll see something like this.

Hero Search Component

Final code review

Your app should look like this live example / download example.

Here are the code files discussed on this page (all in the src/app/ folder).

HeroService, InMemoryDataService, AppModule

hero.service.ts

in-memory-data.service.ts

app.module.ts

content\_copy

import { Injectable } from '@angular/core';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable, of } from 'rxjs';

import { catchError, map, tap } from 'rxjs/operators';

import { Hero } from './hero';

import { MessageService } from './message.service';

const httpOptions = {

headers: new HttpHeaders({ 'Content-Type': 'application/json' })

};

@Injectable({ providedIn: 'root' })

export class HeroService {

private heroesUrl = 'api/heroes'; // URL to web api

constructor(

private http: HttpClient,

private messageService: MessageService) { }

/\*\* GET heroes from the server \*/

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

.pipe(

tap(heroes => this.log('fetched heroes')),

catchError(this.handleError('getHeroes', []))

);

}

/\*\* GET hero by id. Return `undefined` when id not found \*/

getHeroNo404<Data>(id: number): Observable<Hero> {

const url = `${this.heroesUrl}/?id=${id}`;

return this.http.get<Hero[]>(url)

.pipe(

map(heroes => heroes[0]), // returns a {0|1} element array

tap(h => {

const outcome = h ? `fetched` : `did not find`;

this.log(`${outcome} hero id=${id}`);

}),

catchError(this.handleError<Hero>(`getHero id=${id}`))

);

}

/\*\* GET hero by id. Will 404 if id not found \*/

getHero(id: number): Observable<Hero> {

const url = `${this.heroesUrl}/${id}`;

return this.http.get<Hero>(url).pipe(

tap(\_ => this.log(`fetched hero id=${id}`)),

catchError(this.handleError<Hero>(`getHero id=${id}`))

);

}

/\* GET heroes whose name contains search term \*/

searchHeroes(term: string): Observable<Hero[]> {

if (!term.trim()) {

// if not search term, return empty hero array.

return of([]);

}

return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(

tap(\_ => this.log(`found heroes matching "${term}"`)),

catchError(this.handleError<Hero[]>('searchHeroes', []))

);

}

//////// Save methods //////////

/\*\* POST: add a new hero to the server \*/

addHero (hero: Hero): Observable<Hero> {

return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(

tap((hero: Hero) => this.log(`added hero w/ id=${hero.id}`)),

catchError(this.handleError<Hero>('addHero'))

);

}

/\*\* DELETE: delete the hero from the server \*/

deleteHero (hero: Hero | number): Observable<Hero> {

const id = typeof hero === 'number' ? hero : hero.id;

const url = `${this.heroesUrl}/${id}`;

return this.http.delete<Hero>(url, httpOptions).pipe(

tap(\_ => this.log(`deleted hero id=${id}`)),

catchError(this.handleError<Hero>('deleteHero'))

);

}

/\*\* PUT: update the hero on the server \*/

updateHero (hero: Hero): Observable<any> {

return this.http.put(this.heroesUrl, hero, httpOptions).pipe(

tap(\_ => this.log(`updated hero id=${hero.id}`)),

catchError(this.handleError<any>('updateHero'))

);

}

/\*\*

\* Handle Http operation that failed.

\* Let the app continue.

\* @param operation - name of the operation that failed

\* @param result - optional value to return as the observable result

\*/

private handleError<T> (operation = 'operation', result?: T) {

return (error: any): Observable<T> => {

// TODO: send the error to remote logging infrastructure

console.error(error); // log to console instead

// TODO: better job of transforming error for user consumption

this.log(`${operation} failed: ${error.message}`);

// Let the app keep running by returning an empty result.

return of(result as T);

};

}

/\*\* Log a HeroService message with the MessageService \*/

private log(message: string) {

this.messageService.add(`HeroService: ${message}`);

}

}

HeroesComponent

heroes/heroes.component.html

heroes/heroes.component.ts

heroes/heroes.component.css

content\_copy

<h2>My Heroes</h2>

<div>

<label>Hero name:

<input #heroName />

</label>

<!-- (click) passes input value to add() and then clears the input -->

<button (click)="add(heroName.value); heroName.value=''">

add

</button>

</div>

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<a routerLink="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</a>

<button class="delete" title="delete hero"

(click)="delete(hero)">x</button>

</li>

</ul>

HeroDetailComponent

hero-detail/hero-detail.component.html

hero-detail/hero-detail.component.ts

content\_copy

<div \*ngIf="hero">

<h2>{{hero.name | uppercase}} Details</h2>

<div><span>id: </span>{{hero.id}}</div>

<div>

<label>name:

<input [(ngModel)]="hero.name" placeholder="name"/>

</label>

</div>

<button (click)="goBack()">go back</button>

<button (click)="save()">save</button>

</div>

HeroSearchComponent

hero-search/hero-search.component.html

hero-search/hero-search.component.ts

hero-search/hero-search.component.css

content\_copy

<div id="search-component">

<h4>Hero Search</h4>

<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

<ul class="search-result">

<li \*ngFor="let hero of heroes$ | async" >

<a routerLink="/detail/{{hero.id}}">

{{hero.name}}

</a>

</li>

</ul>

</div>

Summary

You're at the end of your journey, and you've accomplished a lot.

You added the necessary dependencies to use HTTP in the app.

You refactored HeroService to load heroes from a web API.

You extended HeroService to support post(), put(), and delete() methods.

You updated the components to allow adding, editing, and deleting of heroes.

You configured an in-memory web API.

You learned how to use observables.

This concludes the "Tour of Heroes" tutorial. You're ready to learn more about Angular development in the fundamentals section, starting with the Architecture guide.